Documenting Software Architecture: Documenting Behavior

Felix Bachmann

Len Bass

Paul Clements

David Garlan

James Ivers

Reed Little

Robert Nord

Judith Stafford

January 2002

Documenting Software Architecture: Documenting Behavior

Felix Bachmann
Len Bass
Paul Clements
David Garlan
James Ivers
Reed Little
Robert Nord
Judith Stafford

This report was prepared for the

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER
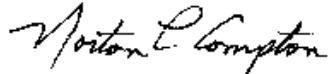
Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

Table of Contents

List of Figures

List of Tables

Abstract

This report represents another milestone of a work in progress: a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively titled *Documenting Software Architectures*, will be published in early 2002 by Addison-Wesley as part of the Software Engineering Institute (SEI) Series on Software Engineering.

The book is intended to address a lack of language-independent guidance about how to capture an architecture in a written form that can provide a unified design vision to all of the stakeholders on a development project.

A central precept of the book is that documenting an architecture entails two essential steps: (1) documenting the set of relevant views of that architecture, and then completing the picture by (2) documenting information that transcends any single view. The book's audience is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

This technical note describes ways to document an important but often overlooked aspect of software architecture: the behavior of systems, subsystems, and components.

1   Introduction

This report represents another milestone of a work in progress: a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively titled *Documenting Software Architectures*[1], will be published in early 2002 by Addison-Wesley as part of the Software Engineering Institute (SEI) Series on Software Engineering. Since this report is a snapshot of current work, the material described here may change before the handbook is published.

The book is intended to address a lack of language-independent guidance about how to capture an architecture in a written form that can provide a unified design vision to all of the stakeholders on a development project.

A central precept of the book is that documenting an architecture entails two essential steps: 1) documenting the set of relevant views of that architecture and then completing the picture by 2) documenting information that transcends any single view. The book's audience is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

Two previous reports laid out our approach and organization for the complete book and provided self-contained previews of individual chapters. The first provided guidance for one of the most commonly used architectural views: the layer diagram [Bachmann et al. 00]. The second laid out a structure for a comprehensive architecture documentation package [Bachmann et al. 01].

This technical note describes ways to document an important but often overlooked aspect of software architecture: the behavior of systems, subsystems, and components.

---

[1]   A previous working title was *Software Architecture Documentation in Practice*.

2 Beyond Structure

The classical approach for architecture documentation, and one which we endorse, is organizing the architectural documentation as a collection of architectural views. Most people describe views principally in terms of the structural relationships among the views' elements. However, architecture extends beyond structure. Without taking into account how the elements behave when connected to each other, there can be no assurance that the system will work as intended. Achieving such assurances before the system has been fully implemented is a major goal of paying attention to its architecture. Element behavior, therefore, is an essential part of architecture and therefore of architecture documentation.

This report focuses on the value of, and techniques for, documenting behavioral aspects of the interactions among system elements. Documenting behavior is a way to add more semantic detail to elements and their interactions that have time-related characteristics.

Documenting the behavioral aspects of a view requires that a "time line" of some sort be provided along with structural information. Structural relationships provide a view of the system that reflects all potential interactions; few of which will actually be active at any given instant during system execution. It is the system behavior that describes how element interactions may affect one another at any point in time or when in a given system state. Every view can have an associated description that documents the behavior of the elements and relationships of that view.

Some system attributes can be analyzed entirely using a system's structural description. For example, the existence of anomalies, such as required inputs for which there is no source available, can be detected in a manner similar to the def-use analysis performed by compilers. However, reasoning about properties such as a system's potential to deadlock or a system's ability to complete a task in the required amount of time requires that the architectural description contain information about both the behavior of the elements and constraints on the interactions among them. A behavioral description adds information that reveals

- ordering of interactions among the elements
- opportunities for concurrency
- time dependencies of interactions (at a specific time or after a period of time)

Interaction diagrams or statecharts as defined by the Unified Modeling Language (UML) are examples of behavioral descriptions.

The remainder of this report provides guidance as to what aspects of behavior to document and how this documentation is used during the earliest phases of system development. In addition, we provide overviews and pointers to languages, methods, and tools that are available to help practitioners document system behavior.

3   Where to Document Behavior

Architects document behavior to show how an element behaves when stimulated in a particular way, or to show how an ensemble of elements (up to and including the whole system) react with each other. In an architectural documentation package, behavior can be shown in a number of places, depending on what exactly is being shown:

- In a view's supporting documentation:
    - Behavior has its own section in the element catalog. Here, the behavior of the element is documented.
    - Behavior can be part of an element's interface documentation. The semantics of a resource on an element's interface can include the element's (externally visible) behavior that occurs as a result of invoking the resource. Or, in the "Usage Guide" section of an interface document, behavior can be used to explain the effects of a particular usage, that is, a particular sequence of resources utilized. Finally, the architect may choose to specify behavior as part of the implementation notes for an interface, to constrain the developers to implement the interface in a particular fashion.
    - Behavior can be used to fill in the "Design Background" section, which includes the results of the analysis. Behavior is often a basis for analysis, and the behaviors that were used to analyze the system for correctness or other quality attributes can be recorded here.
- In the documentation that applies across views, the rationale for why the architecture satisfies its requirements can include behavioral specifications as part of the architect's justification.

4   Why Document Behavior?

The documentation of system behavior is used for system analysis and for communication among stakeholders during system-development activities.

The types of analysis you perform and the extent to which you check the quality attributes of your system will be based on the type of system that you are developing. It is generally a good idea to do some type of tradeoff analysis to determine the costs and risks involved with applying certain types of architectural analysis techniques. For any system it is a good idea to identify and simulate a set of requirements-based scenarios. If you are developing a safety-critical system, the application of more expensive, formal analysis techniques (such as model checking) is justified in order to identify possible design flaws that could lead to safety-related failures.

4.1   System Analysis

If you have a behavioral description, you can reason about the system's completeness, correctness, and quality attributes.

Once the structure of an architectural view has been identified and the interactions among elements have been constrained, it is time to take a look at whether the proposed system is going to be able to do its job the way in which it should. This is your opportunity to reason about both the completeness and the correctness of the architecture. It is possible to simulate the behavior of the proposed system in order to reason about the architecture's ability to support system requirements both in terms of whether it supports the range of functionality that it is supposed to and also to determine whether it will be able to perform its functions in a way that is consistent with its requirements.

Documenting system behavior provides support for exploring the quality attributes of a system very early in the development process. There are some existing techniques and some still in development that you can use to predict the architecture's ability to support the production of a system that exhibits specific measures related to properties such as aspects of performance, reliability, and modifiability.

Architecture-based simulation is similar in nature to testing an implementation in that a simulation is based on a specific use of the system under specific conditions and with the expectation of a certain outcome. Typically, a developer will identify a set of scenarios based on the system requirements. These scenarios are similar to test cases in that they identify the stimulus of an activity and the assumptions about the environment in which the system is running, and describe the expected simulation results. These scenarios are played out against a description of the system that supports relating system elements and the constraints on their interactions. The results of "running the architecture" are checked against the expected behavior. A variety of notations for documenting the results of system simulation are discussed further in Section 6.2.

Whereas simulation looks at a set of special cases, system-wide techniques for analyzing the architecture evaluate the overall system. These include analysis techniques for dependence, deadlock, safety, and schedulability. These techniques require information about the behavior of the system and its constituent elements in order to compute the property values. The analysis of inter- and intra-element dependencies has many applications in the evaluation of system-quality attributes. Dependence analysis is used as a supporting analysis to help evaluate quality attributes such as performance and modifiability.

Compositional-reasoning techniques that are available today, and those that being developed in research laboratories, require information about both the internal behavior of system elements and interactions among them. This information is stated either as a summarization of the actual behavior of existing elements or as derived requirements that the implemented element must satisfy in order to assure the validity of analysis results. In either case you will need to document internal element behavior in some way if you are to reap the benefits of early system analysis.

4.2   Driving Development Activities

Behavioral documentation plays a part in architecture's role as a vehicle for communication among stakeholders during system-development activities. The activities associated with architectural documentation produce confidence that the system will be able to achieve its goals. Many decisions about the structure of the system were made and documented based on the perspectives of a variety of stakeholders in the system's development. The process of designing the architecture helps the architects to develop an understanding of the internal behavior of system elements as well as an understanding of gross system structure. This understanding can be captured in various types of behavioral documentation and later used to more precisely specify inter-element communication and intra-element behavior.

System decomposition results in the identification of sets of sub-elements and the definition of both the structure and the interactions among the elements of a given set in a way that supports the required behavior of the parent element. In fact, the behavior defined for the parent element has important influence on the structure of its decomposition. As an example, consider an assignment to design a gateway. The responsibility of a gateway is to receive messages from one protocol and translate them into another protocol, and then to send them out again. Unfortunately for many protocols, this translation cannot be done message by message. A set of messages from one protocol may translate into a single message of the other protocol, or the content of a translated message may depend on earlier messages received. The specified behavior for the gateway describes which sequence of messages would lead to a translated message and which information needs to be kept in order to produce the appropriate message content to be sent. This behavior will likely influence the decomposition in a way that reflects the fact that some elements have the responsibility of dealing with specific sequences of incoming messages and that other elements have the responsibility of storing the required information.

Implementing a system using a defined architecture is a continuous process of decomposition in smaller and more detailed elements by defining the system's structure and behavior until it

is possible to describe the behavior in a programming language. Therefore the behavioral description of the architecture, as well as the structural description, is important input for the implementation process.

Additionally, you might want to use simulation during the development of the system. Stimulus-oriented diagrams (such as sequence diagrams) offer a notation for documenting the results of applying scenarios to a set of elements. Such simulation enables developers to gain early confidence that the system under development will actually fulfill its requirements. Simulation may even convince management that the developers are doing great things. In order to use simulation, a behavioral description of the system or its parts is required. The scenarios used for this purpose can later be used to develop test cases to be applied during integration testing.

5   What to Document

As mentioned above, a behavioral description supports exploring the range of possible orders of interactions, opportunities for concurrency, and time-based interaction dependencies among system elements. In this section we provide guidance as to what types of things you will want to document in order to reap these benefits.

The exact nature of what to model depends on the type of system that is being designed. For example, if the system is a real-time embedded system, you will need to say a lot about timing properties and the ordering of events; whereas, in a banking system you will want to say more about the sequencing of events (e.g., atomic transactions and roll-back procedures). Initially you want to talk about the elements and how they interact, rather than the details of how input data is transformed into outputs. It may be useful to also say something about the constraints on the transformational behavior within elements, in as much as that behavior affects the global behavior of the system.

At a minimum, you will want to model the stimulation of actions and the transfer of information from one element to another. In addition, you may want to model time-related and ordering constraints on these interactions. If correct behavior depends on restrictions as to the order in which actions must occur or as to combinations of actions that must have occurred before a certain action can be taken, then these things must be documented. The more information that is available and made explicit about the constraints on interactions, the more precise the analysis of system behavior can be, and the more likely it will be that the implementation exhibits the same qualities as those predicted during design.

5.1   Types of Communication

Looking at a structural diagram that depicts two interrelated elements, the first questions documentation users ask are "What does the line interconnecting the elements mean?" and "Is it showing the flow of data or control?" A behavioral diagram provides a place to describe these aspects of the transfer of information and the stimulation of actions from one element to another. Table 1 shows examples of these. Data is some kind of structured information that may be communicated through shared files and objects. One element may stimulate another to signal that some task is completed or that a service is required. A combination of the two is possible, as is the case when one element stimulates another to deliver data or when information is passed in messages or as event parameters.

Table 1:   Types of Communication

	synchronous	**asynchronous**
data	N/A	database, shared memory
stimulation	procedure call without data parameters	interrupt
both	procedure call, remote procedure call (RPC)	message, events with parameters

In addition to the above, you may want to describe constraints on the interaction between elements in the form of synchronous or asynchronous communication. An RPC is an example of synchronous communication. The sender and receiver know about each other and synchronize in order to communicate. Messaging is an example of asynchronous communication. The sender does not concern itself with the state of the receiver when sending a message or posting an event. In fact, the sender and receiver may not be aware of the identity of each other. Consider telephone and email as examples of these types of communication. If you make a phone call to someone, they have to be at their phone in order for it to achieve its full purpose. That is synchronous communication. If you send an email message and go on to other business, perhaps without concern for a response, then it is asynchronous.

5.2   Constraints on Ordering

In the case of synchronous communication, you probably want to say more than that there is two-way communication. For instance, you may want to state which element initiated the communication and which element will terminate it; you may want to say whether the target of the original message will need to employ the assistance of other elements before it can respond to the original request. Decisions about the level of detail at which you describe a conversation depend upon which types of information you want to get out of the specification. For instance, if you are interested in performance analysis, it is important to know that an element will reach a point in its calculation where it requires additional input, since the length of the total calculation depends not only on the internal calculation, but also on the delay associated with waiting for required inputs.

You will probably want to be more specific about certain aspects of the way an element reacts to its inputs. You may want to note whether an element requires all or just some of its inputs to be present before it begins calculating. You may want to say whether it can provide intermediate outputs or only final outputs. If a specific set of events must take place before an element's action is enabled, that should be specified, as should the circumstances in which a set of events or element interactions will be triggered or the environment in which an output of an element is useful. These types of constraints on interactions provide information that is useful for analyzing the design for functional correctness as well as for extra, functional properties.

5.3   Clock-Triggered Stimulation

If activities are specified to take place at specific times or after certain intervals of time, some notion of time will need to be introduced into your documentation. Using two types of clocks is helpful for this purpose. One clock measures calendar time to whatever precision is required for the type of system under construction. This clock allows you to specify that certain things are to happen at certain times of the day or month. For instance, you may want to specify some behavior differently for weekends and holidays. The other clock counts tics or some other, perhaps more precisely specified, measure of time. This clock allows you to specify periodic actions, for example, directions to check every five minutes and determine how many people are logged on to the system. While it is clearly possible to compute one clock from the other, it is simpler to use both mechanisms when creating your architectural documentation, since these are two different ways of thinking about time.

6 How to Document Behavior: Notations and Languages

Any notation that supports documenting system behavior must include constructs for describing sequences of interactions. Since a sequence is an ordering in time, it should be possible to show time-based dependencies. Sequences of interactions are displayed as a set of stimuli and the triggered activities ordered into a sequence by some means (e.g., a line, numbering, ordering, from top to bottom). Examples of stimuli are the passage of time and the arrival of an event. Examples of activities are compute and wait. Notation that shows time as a point (e.g., time-out) and time as an interval (e.g., wait for 10 seconds) are normally also provided. As a description of behavior implicitly refers to structure and uses structure, the structural elements can be part of the notation. Therefore in most behavior documentation, you can find representations of

- stimulus and activity
- ordering of interactions
- structural elements with some relationships to which the behavior maps

Two different groups of behavioral documentation are available, and the notations to support the documentation of behavior tend to fall into one of two corresponding camps:

- **static views**. One type of documentation, often state based, shows the complete behavior of a structural element or set of elements. This is referred to as a *static view* of behavior, because it is possible to infer all possible traces through a system given this type of documentation. Static behavioral documentation supports the description of alternatives and repetitions to provide the opportunity of following different paths through a system depending on runtime values. With this type of documentation, it is possible to infer the behavior of the elements in any possible case (arrival of any possible stimulus). Therefore, this type of documentation should be chosen when a complete behavior description is required, as is the case for performing a simulation or when applying static-analysis techniques.
- **traces**. Another type of documentation shows *traces* (e.g., interaction diagrams) through the structural elements. Those traces are only complete with regard to what happens in a system in case a specific stimulus arrives. Trace descriptions are by no means complete behavioral descriptions of a system. On the other hand, the union of all possible traces would generate a complete behavioral description. Trace descriptions are easier to design and to communicate because they have a narrow focus. Consequently, if the goal is to understand the system or to analyze a difficult situation that the system has to deal with, a trace-oriented description for the behavior is the first choice.

There are many notations available for both types of behavioral documentation. The differences between these methods lay in the emphasis that is put on certain aspects of the behavior (stimulus, activity, ordering, elements). There is also a difference in how much detail can be described. In a real-time environment where the timing behavior is important, you might want to describe not only the ordering of stimuli/activity in time but also the amount of time con-

sumed by an activity. This could be done, for example, by having textural annotations on activities or by having an underlying "time grid."

In the sections that follow, we provide cursory overviews of several notations within each of these categories. The discussions are intended to provide a flavor of the particular notations and to motivate their use. There are many ways in which the diagrams we present in this section may be used together to support the design process. One possible set of representations that uses the strengths of several different notations for describing activities during the design process of a system is shown in Figure 1. Functional requirements are represented as use cases, which help to clarify the understanding of the requirements and the system boundaries. Use-case maps (UCMs) describe how the use cases work their way through the elements of a system and are used as the basis for defining the messages between the elements, using one of the message-interaction diagrams such as sequence diagrams, collaboration diagrams, or message-sequence charts (MSCs). Once the message interface between the elements is well understood, a static behavioral model may be used to describe the internal behavior of the elements. This model might be a state-based formalism (such as a statechart, ROOMchart, or SDL flowchart) or a formalism based on pre- and post-conditions (such as the Z language).

| Use Cases | Use-Case Maps | Interaction Diagrams | Statecharts |
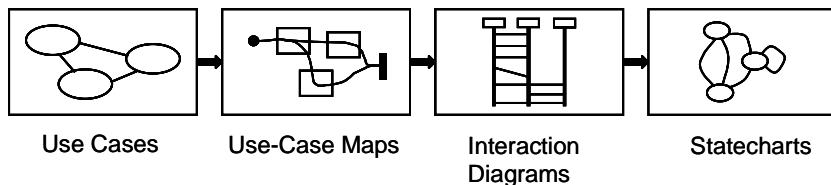
Figure 1:   Possible Usage of Different Behavioral Descriptions

6.1   Static Behavioral Modeling

Static behavioral modeling shows the complete behavior of a structural element or set of elements. It is possible to infer all of the possible traces through a system given this type of documentation. The state-machine formalism is a good candidate for representing the behavior of architectural elements, because each state is an abstraction of all the possible histories that could lead to that state. Once in a state, it doesn't matter how the system got there, only that it is there; it will react to the occurrence of a given event in the same way regardless of the system's particular history at the time the event occurs. Notations are available that allow you also to describe the internal behavior of elements in terms of finite state machines and element-to-element interactions in terms of interprocess communication of various types.   These notations allow you to overlay a structural description of the system's elements with constraints on the interactions and timed reactions to both internal and environmental stimuli.

In this section we describe two state-based notations: statecharts, which are extensions to the basic notion of finite state machines, and ROOMcharts, which further extend the notion to address the needs of object-oriented descriptions of the behavior of real-time systems. We also describe the Specification and Description Language (SDL) and the Z language.

Although other notations are available, we have chosen these because they allow us to describe the basic concepts of documenting behavior in forms that capture the essence of what you wish to convey to system stakeholders. They are also used as base representations in the tools that you are most likely to encounter. Each notation has been incorporated into one or more development environments that allow you to design, simulate, and analyze your system early in the development process.

6.1.1   Statecharts

A statechart is a formalism that was developed by David Harel in the 1980s for describing reactive systems. Statecharts are powerful graphical notations that allow you to trace the behavior of your system given specific inputs. Statecharts add a number of useful extensions to traditional state diagrams, such as the nesting of states and orthogonal regions (AND states). These provide the expressive power to model abstraction and concurrency.

A limitation of finite-state-machine representations is that there is no notion of depth. Statecharts extend the finite-state-machine formalism to support the description of the transformations within a state in terms of nested states. The outer state is called the *superstate* and inner states are referred to as *substates*. When the superstate is entered, all substates are initiated at their respective default start state, and they remain active until the superstate is exited. A state runs when all entry conditions are fulfilled. The behavior of any substate can be expanded if desired. Substates can be related either by sequence (i.e., one state leads to another depending on the occurrence of events) or by concurrency (i.e., states are in orthogonal regions and are activated upon entry to the superstate).

Statecharts have their limitations. Several simplifying assumptions are incorporated into the statechart model. Among these is the assumption that all transitions take zero time. This allows a set of transitions within a substate to replace a single transition at the superstate level. As an example, the transition from entering a state to exiting it is taken to be zero. However if we expanded the state, we might see that there are several transitions within it. Clearly each transition takes time, but this fact is abstracted away in statecharts. Additionally, statecharts do not provide any built-in support for modeling protocols; state transitions are instantaneous and reliable. These simplifying assumptions allow you to record and analyze your system before many design decisions are made. However as you refine your knowledge of the system, you will want to create more precise descriptions.

The example statechart shown in Figure 2 illustrates the states that some of the JavaPhone™ objects (Call, Connection, and Terminal Connection) can be in when a phone connection is established and disconnected. This statechart contains important states and transitions but is by no means complete.

TM   Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.
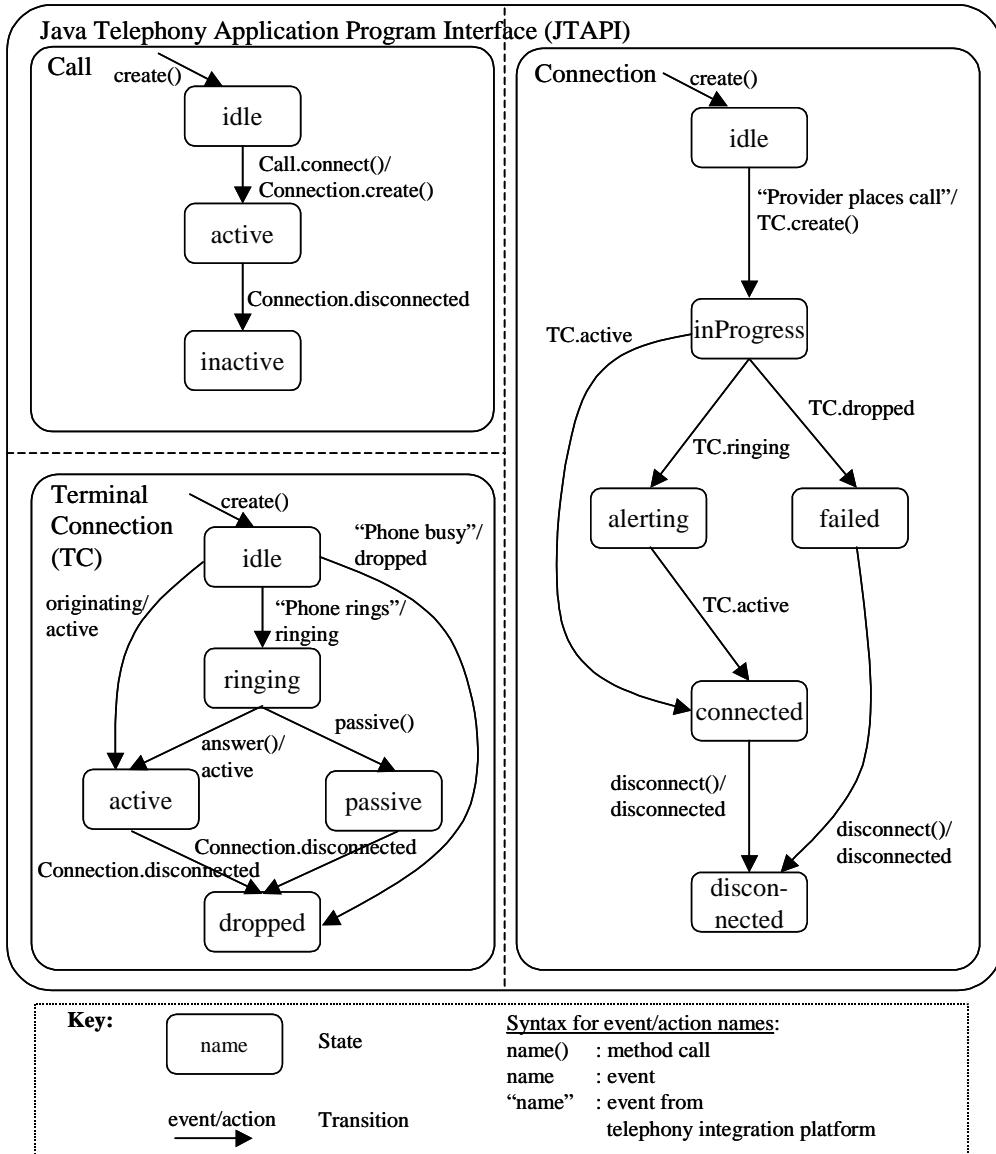
Figure 2:   Statechart Representation of JavaPhone

Sequence is represented by a single-headed arrow leading from the source state to the target state and is annotated with a pair consisting of the possibly parameterized event that causes the transformation, separated by a slash from any events generated along with the state transition. Thus a transformation sets in motion the change in state of one system element and the triggering of transformations in others. Concurrency is represented by grouping sets of states into a superstate, where the states are separated by dotted lines and where there are no arcs between the states.

Here, the JavaPhone superstate contains the substates: Call, Connection, and Terminal Connection. The default start for each substate is depicted by an arrow that has no source state. At the beginning, Call is in the idle state. As soon as the connect() event arrives, a Connection is created, which transitions into the idle state. From there commands are exchanged with the telecommunication platform and Terminal Connections are created. Terminal Connections receive events from the telecommunication platform, which lead to state changes. Those changes trigger state changes in the Connection, which in turn trigger state changes in the Call.

6.1.2   ROOMcharts

Real-time object-oriented modeling (ROOM) is an approach to developing software that is particularly designed to support the use of object-oriented design techniques to aid in the development of real-time systems that are driven by scheduling demands. Because ROOM is an object-oriented approach, it supports the use of data abstraction, encapsulation, and inheritance. The primary objects in ROOM descriptions are actors that communicate by exchanging messages. The behavior associated with an actor is documented as a hierarchical state machine and is incorporated into a ROOMchart.

A ROOMchart is a graphical notation that is a close cousin to the statechart. The concepts in ROOMcharts are very close to commonly used, object-oriented constructs, thus allowing a smooth transition from the high-level design associated with an architectural description down to the detailed description of the system's implementation. The desire to include this feature is one of the reasons that statecharts were not incorporated directly into ROOM. The developers of ROOM wanted to support describing the details of protocols and scheduling. Supplementing statecharts in this way made it necessary to exclude other features. The most notable exclusion is direct support for documenting composite AND states. The lack of this feature does not preclude the representation of orthogonality however. Other features of ROOM can be used to achieve the same goal but with more effort required. One additional feature offered in ROOM is support for the modeling of major concepts associated with object-oriented languages such as inheritance and encapsulation. Behavioral inheritance is also included; thus all features of behavior can be inherited among related actor classes.

The developers of ROOM were particularly interested in providing a way to support developing a system in pieces at various levels of detail at the same time. The ROOM modeling environment supports execution of the model and thereby supports simulation of the architecture. Executable ROOMcharts run on a virtual machine provided by the ROOM environment. The virtual machine provides a set of predefined services; others can be defined by users. Among the predefined, interdependent services are timing, processing, and communication services. The timing service supports both types of time mentioned in "Clock-Triggered Stimulation" on page 8.

This capability to create more precise descriptions required more effort from the modeler and made it necessary to trade off some of the expressive power of statecharts.

6.1.3   Specification and Description Language (SDL)

The SDL is an object-oriented, formal language that was defined by the International Telecommunication Union (ITU). This language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. The most common application is in the telephony area.

The accessible SDL can be used in an environment that is constructed of tools that support the documentation, analysis, and generation of systems. Its design was driven by the requirements

of developing communication systems; thus it will be particularly useful to you if that is the type of system you are developing. The strength of the SDL is in describing what happens within a system. If the focus is on the interaction between systems, a message-oriented representation such as an MSC is more suitable. SDL specifications are often used in combination with MSCs (discussed later in this report) to explore the behavioral properties of a system.

The SDL uses a finite-state-machine formalism at its core to model behavior. The notation focuses on the transition between states rather than the states themselves, as was the case in statecharts and ROOMcharts. Constructs for describing the hierarchical structure and the inter-element behavior enhance the capability for modeling large-scale systems.

In the SDL, structure is described in terms of a hierarchy of blocks that is eventually refined into sets of processes as shown in Figure 3. The flow of data and stimulation among blocks
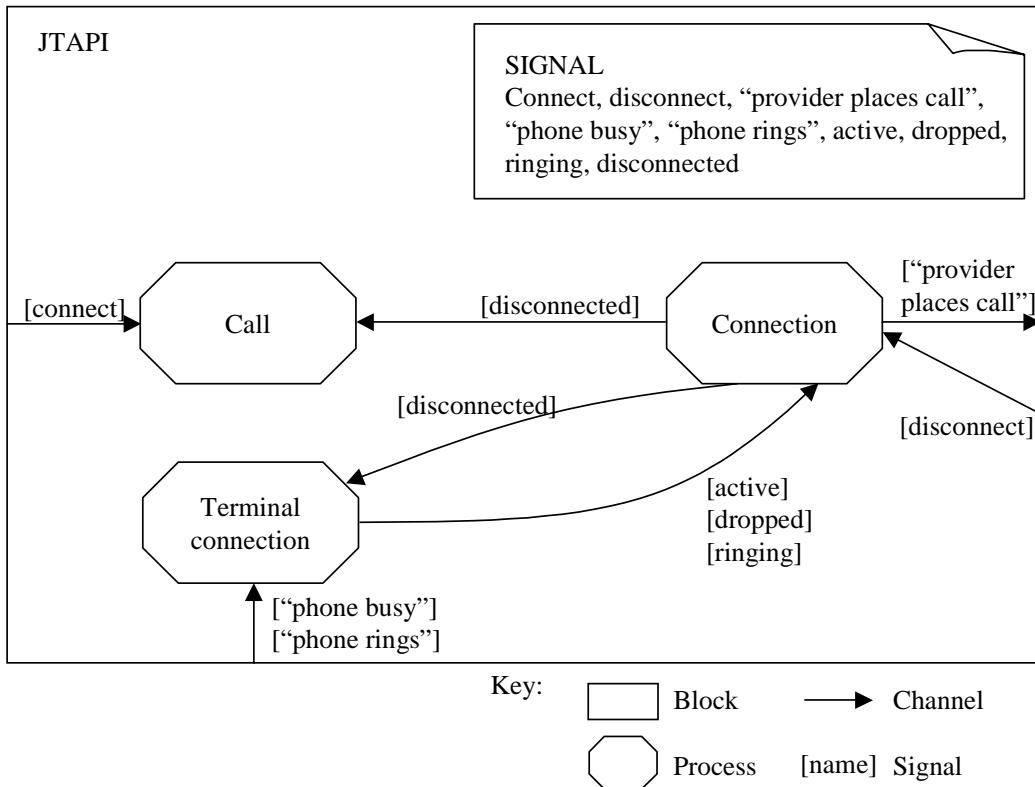
Figure 3: *Hierarchical Structure in the SDL*

The structure of a system is decomposed into a hierarchy of named blocks. Blocks are composed of either blocks or processes, but not combinations of both.

and processes is described as signals that travel over named channels. *Signals* are the means of communication between blocks and processes. Communication is asynchronous and specified textually as an annotation attached to a communication channel. Signals are visible to other blocks/processes at lower levels in the hierarchy, rather than enclosing blocks or other blocks at the same level.

The internal behavior of a process is described in the finite-state-machine formalism using the flowchart notation. Processes run concurrently and independently; concurrent processes have no knowledge of each other's state. Processes can be instantiated at start-up or while the system is running. The SDL provides a rich set of flowchart symbols, a few of which are used in Figure 4 to describe a simple process that checks a user ID for validity.
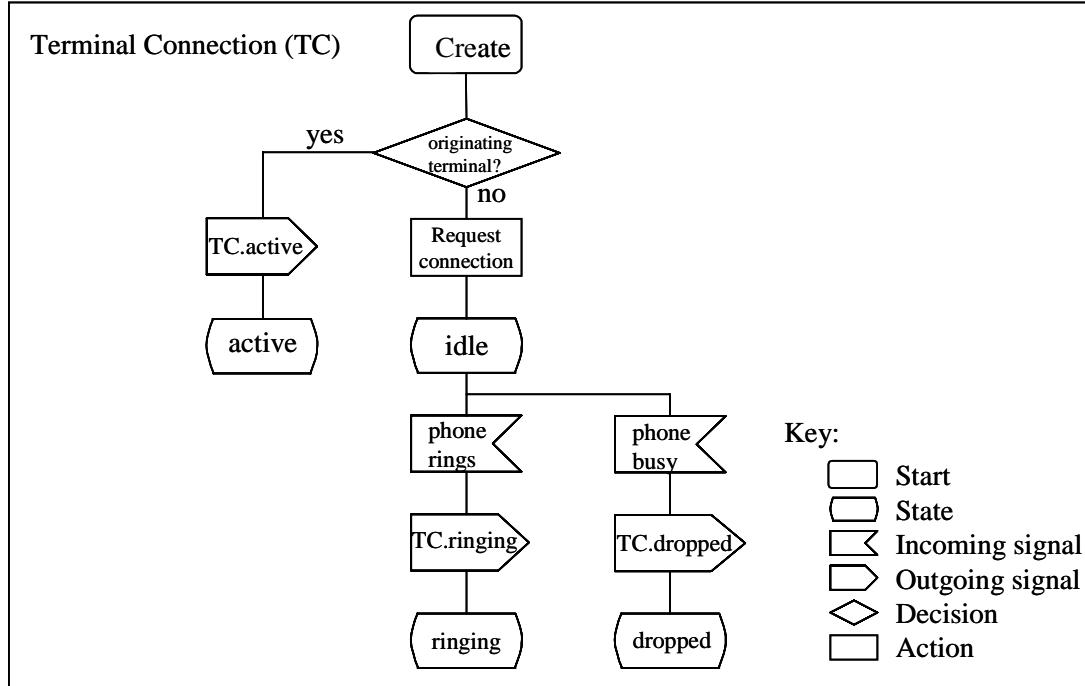
Figure 4: *Intra-Process Behavior in an SDL Flowchart*

> The various shapes represent specific aspects of behavior including changing states, receiving input and sending output, making decisions, and so on, and the arrows represent the flow from one activity to the next in the direction of the arrow.

The SDL supports user-defined data types and provides several predefined types (integer, real, natural, Boolean, character, charstring, PId, duration, and time) that have expected meanings. Variables, user-defined data types, and constant data values can be declared.

The hierarchy of blocks provides a structural view of the system, while the flow among the blocks and processes combined with process flowcharts describes system behavior. Once these aspects have been documented, it is possible to simulate the system and observe control and data flow through the system as signals pass from block to block and into processes where they move through the flowchart representation of process behavior. This type of simulation allows you to visibly check how your system will react to various stimuli.

6.1.4　Z Language

Z, pronounced "zed," is a mathematical language based on predicate logic and set theory. The goal for the Z language was that it be a rigorously defined language that would support the formal description of a system's abstract properties. The Z language focuses on data and its transformations. Systems are specified as sets of *schemas*. Schemas are combined using the schema calculus to create a complete behavior. The schema calculus allows type checking. Tools are available for performing type checking as well as other types of behavioral analysis.

Schemas allow the designer and other users of the specification to focus concern on one small aspect of the system at a time. Simplicity is achieved by breaking a problem into small pieces that can be reasoned about in isolation. A schema is a description of some unit of functionality in terms of a set of variables and the pre- and post-conditions of the system state associated with those variables. This allows a great deal of design freedom in that behavior is specified independently of how tasks are performed. The Z language supports a compositional approach to development and thereby provides the benefit of increased tractability when designing large systems. The Z language is particularly useful when you desire to prove properties based on the specification, as is the case when building safety-critical systems. In addition, an array of commercial tools is available to support developing systems based on the Z language. These are some of the reasons that many practitioners who are experienced in the use of the language consider it to be an invaluable tool. However, because it includes a large set of symbols and its expressions are written in terms of predicate logic, it is difficult for some designers to warm up to it.

The *ScheduleClass* schema shown in Figure 5 defines what it means to add a class to a schedule and provides only the flavor of a Z schema. There are many other constructs available for specifying more complex types of relationships. A description of the schema calculus is beyond the scope of this presentation as are the details of Z type checking and other aspects of the specification language. As mentioned earlier, there are many references available if you are interested in using Z.

6.2　Trace-Oriented Representations

Trace-oriented representations consist of sequences of activities or interactions that describe the system's response to a specific stimulus. They document the trace of activities through a system described in terms of its structural elements and their interactions. Although it is conceivable to describe all possible traces through a set of elements to generate the equivalent of a static behavior description, it is not the intention of trace-oriented views to do so. This would reduce the benefit of being readily comprehensible due to the resultant loss of focus.

Different techniques emphasize different aspects of behavior:

* Message-oriented techniques focus on describing the message exchange between instances. They show sequences of messages and possibly time dependencies. The basic assumption here is that you will be able to understand and/or build an element if you

```
  ScheduleClass _____

        DSchedule
        class?: CLASS_NAME
        time?: TIME
     _____

        class? ∉ scheduled

        scheduled' = scheduled ∪ {class? Œ time?}
```
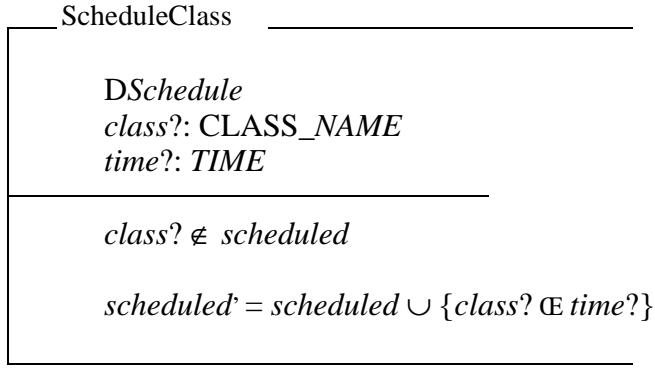
Figure 5:   Example Z Schema

The lines above the center horizontal line are variable definitions. The letter *D* signifies the fact that a schema named Schedule exists and that all of its variables are available to ScheduleClass. The variable names that end in a question mark (*?*) are input variables. The text below the center horizontal line first gives pre-conditions for an operation and then states the promised results of the transformation. The single quotation mark (') attached to the word *scheduled* indicates that the variable it is attached to will be transformed into the result of the expression on the right side of the equals sign (=). In this case the class will be added to the schedule and will be associated with the specified time.

understand which messages arrive at this element and what the reactions in terms of out-going messages have to be. Internal features of the element(s) are hidden.

- Component-oriented techniques focus on describing which behavioral features an element has to have in order to accommodate the system in performing its functions. This normally focuses on describing how the interfaces of the elements interact with each other in order to fulfill the system's functional requirements. Sequences of interactions can be shown, and internal features of the element(s) are hidden.

- Activity-oriented techniques focus on describing which activities have to be performed in order to achieve the purpose of the system. The assumption here is that in order to under-stand what a system (or element) does (or will do), you need to understand the sequence of activities that it entails. Activity-oriented representations may not even show the elements performing those activities. However, it is assumed that there is some means outside of this specific representation technique that allows the assignment of the described activities to elements.

- Flow-oriented techniques focus on describing the sequencing of responsibilities of ele-ments for a specific scenario or trace. This is useful in understanding concurrency and synchronization.

Now, let's look closer at some of the popular, trace-oriented, representation techniques. We will discuss message-oriented techniques (such as sequence diagrams and MSCs) as well as component-oriented techniques (such as collaboration diagrams and a special version of sequence diagrams, the procedural sequence diagram). In addition we show an example of an activity-oriented representation, which is a use-case diagram, and a flow-oriented representa-tion, which is a UCM.

6.2.1　Use-Case Diagrams

Use-case diagrams show how users interact with use cases and how the latter are interrelated. The purpose of a use case is to define a piece of an element's behavior such as a system or its parts as a sequence of activities, without regard to the internal structure of the element. Therefore, a use-case diagram is an activity-oriented representation.

Each use case specifies a service that the element provides to its users (i.e., a specific way of using the element). The service, which is initiated by a user, is a complete sequence of interactions between the users and the element as well as the responses performed by the element (as these responses are perceived from outside of the element). Use cases by themselves cannot be decomposed, but each element of a system can have a use case that specifies its behavior. Therefore, a complete set of use cases for the children elements of a system decomposition builds the basis for the use cases of the parent element.

Use-case diagrams focus on creating a behavioral description that specifies requirements in a more concise way. These diagrams do not really focus on assigning behavior or stimuli to structural elements, although that can be done using other methods such as sequence or collaboration diagrams. Additionally, use-case diagrams do not have a means to document concurrency, although the underlying assumption is that all use cases can be performed independently.

Figure 6 shows an example of a use-case diagram. The top portion shows how phone terminals interact with the "Establish Point-to-Point Connection" use case. Since phone terminals are external to the specified element, they are represented by actors. An actor is a set of roles that external entities assume when interacting with use cases. There may be associations between use cases and actors, meaning that the instances of the use case and the actor communicate with each other. A use-case instance is initiated by a message from an instance of an actor. As a response, the use-case instance performs a sequence of actions as specified by the use case. These actions may include communicating with actor instances besides the initiating one.

Figure 6 also illustrates how use cases can have relationships with each other. An *extend* relationship defines that instances of a use case may be extended with some additional behavior defined in an extending use case. An extension point references one location or a collection of locations in a use case where the latter may be extended. A *generalization* relationship between use cases implies that the child use case contains all the sequences of behavior and extension points that are defined in the parent use case, and that it participates in all the relationships of the parent use case. The child use case may also define new behavior sequences, as well as add behavior into and specialize the existing behavior of the inherited ones. An *include* relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior that is performed by an instance of the base use case.

Normally a use case is described in plain text, but other techniques (such as sequence diagrams or statecharts) can be attached to a use case to describe its behavior in a more formal way.

6.2.2　Use-Case Maps (UCMs)

The UCM notation was developed at Carleton University by Professor Buhr and his team, and it has been used for describing and understanding a wide range of applications since 1992. UCMs concentrate on visualizing execution paths through a set of elements and provide a bird's-eye, path-centric view of system functionalities. UCMs allow dynamic behavior and structures to be represented and evaluated, and improve the reusability level of scenarios. The fairly intuitive notation of UCMs is very useful to communicate how a system works (or is supposed to work), without getting lost in too much detail.
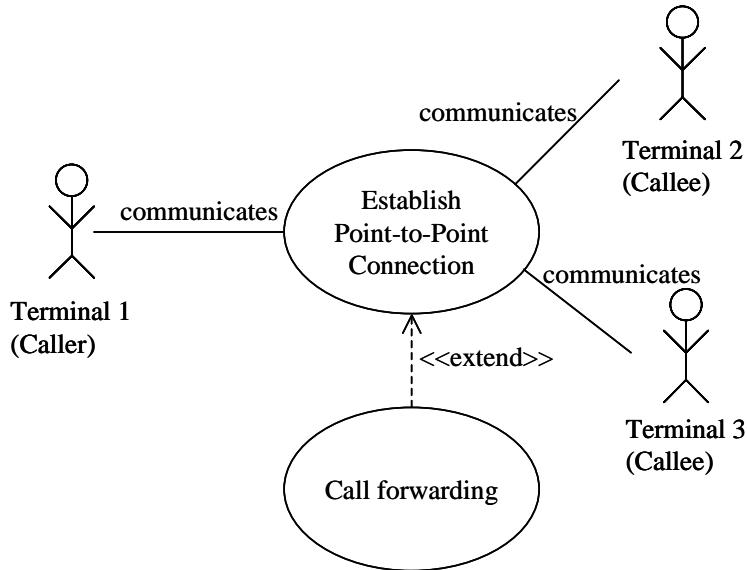

Figure 6:　Use-Case Diagram of JavaPhone

UCMs can be derived from informal requirements or from use cases if they are available. The responsibilities for each actor need to be stated in or inferred from these requirements. For illustration purposes, separate UCMs can be created for individual system functionalities or even for individual scenarios. However, the strength of this notation resides mainly in the integration of scenarios. Therefore, UCMs can be used to illustrate concurrency, such as resource-consumption problems (multiple paths using one element) or possible deadlock situations (two paths in opposite directions through at least two of the same elements).

If you ever followed a discussion of developers who are concerned mainly about concurrency to answer questions like, "Does a component need to be locked?" or "Is there a potential for deadlock?", you may have seen them drawing pictures like the one shown in Figure 7. This type of notation builds the basis for UCMs.

The basic idea of UCMs is captured by the phrase *causal paths cutting across organizational structures*. This means that execution paths describe how elements are ordered according to the responsibilities they carry out. These paths represent scenarios that intend to bridge the gap between functional requirements and a detailed design.The realization of this idea produces a

scalable, lightweight notation, while at the same time covering complexity in an integrated and manageable fashion. The UCM notation aims to link behavior and structure in an explicit and visual way.
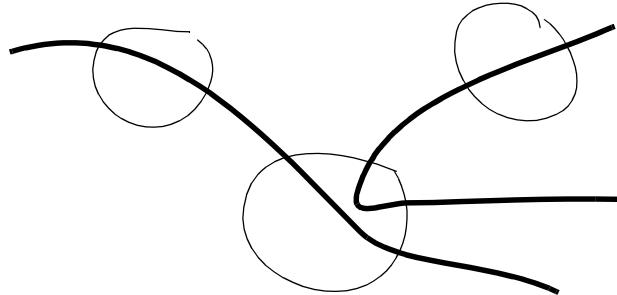

Figure 7:   Sketch of Activities Through Some Components

Like the other representations, UCMs show instances of structural elements. In addition, UCMs have a notation for the "containment" of those elements and thus show a type of relationship that is normally shown in structural descriptions. By doing this, UCMs are easy to understand; it is easy to describe how sub-elements contribute to the system behavior.

When an execution path (a line) enters an element (a box), it states that now this element does its part to achieve the system's functionality. A responsibility assigned to the path within the element's box defines it as a responsibility of this element. The example UCM shown in Figure 8 shows the flow of activities through the elements of a JavaPhone application when a Point-to-Point Connection is established.

The notation includes a means to represent the decomposition of execution paths. This feature allows step-by-step understanding of more and more details of the system. The example includes a decomposition shown by the diamond-shaped symbol. The "Callee service" decomposition is shown in the little UCMs. In this specific case, decomposition is also used to show possible variations. Callee service cannot only be decomposed into a basic call, it also can be decomposed so that the feature "Call forwarding" is added.

The notation for UCMs also includes symbols: for timers (and time-outs); for using data containers; for the interaction between execution paths such as abort; for goals, which are very useful when describing agent-oriented components; and many more.
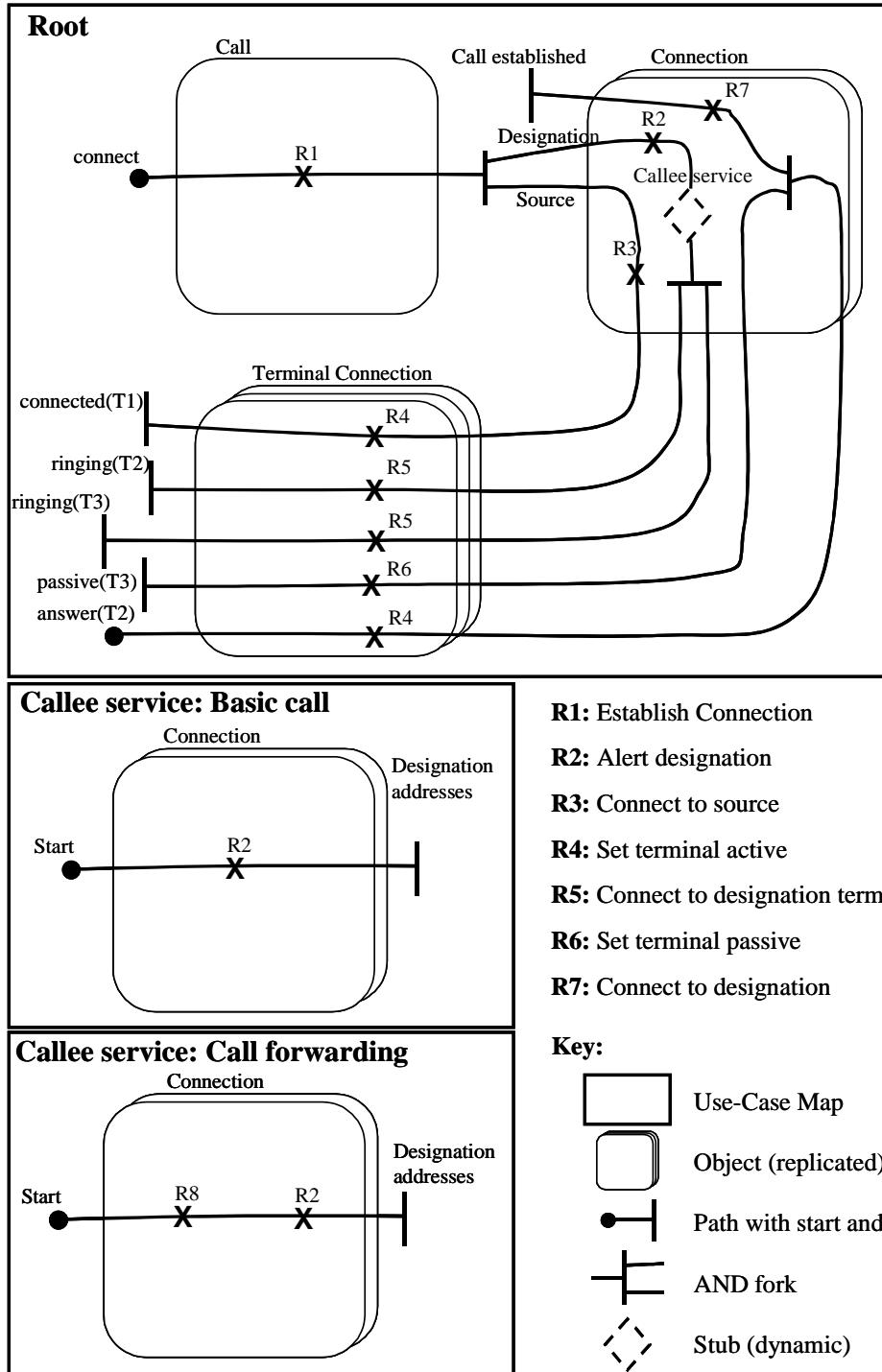
Figure 8:   *"Establish Point-to-Point Connection" UCM*

Execution paths are represented as sets of wiggly lines. Execution paths have a beginning (large dot) and an end (bold straight line). Execution paths can split to show concurrent activities, can have alternative ways, or can join together again. The responsibilities assigned to a path are shown as annotated little crosses on that path. Decomposition and variation are shown as a diamond-shaped symbol in the parent UCM, that has incoming and outgoing execution paths. An assigned child UCM shows what happens in more detail.

6.2.3   Sequence Diagrams

Sequence diagrams document a sequence of stimuli exchanges. A sequence diagram presents a collaboration in terms of instances of elements defined in the structural description with a superimposed interaction and shows that interaction arranged in a time sequence. In particular, a sequence diagram shows the instances participating in the interaction. A sequence diagram has two dimensions:

1. The vertical dimension represents time.
2. The horizontal dimension represents different objects.

In a sequence diagram, associations among the objects are not shown. There is no significance to the horizontal ordering of the objects.

Sequence diagrams support the depiction of dependent interactions nicely, which means that they show which stimulus follows another stimulus. Sequence diagrams are not very explicit in showing concurrency. There might be the assumption that the different sequences of interaction shown in different diagrams actually can be performed independently of each other. If that is the intention when documenting behavior using sequence diagrams, it should be documented somewhere. It definitely is not documented within a sequence diagram, which shows instances as concurrent units; they run in parallel. However, no assumptions can be made about ordering or concurrency when a sequence diagram depicts an instance sending messages at the "same time" to different instances or conversely receiving multiple stimuli at the "same time."

A component-oriented style of sequence diagram is the procedural sequence diagram. This style of diagram focuses on the interface interactions of elements and is more suitable to show concurrency, because it has some means to show flow control, such as decisions and loops.

Figure 9 shows an example sequence diagram.
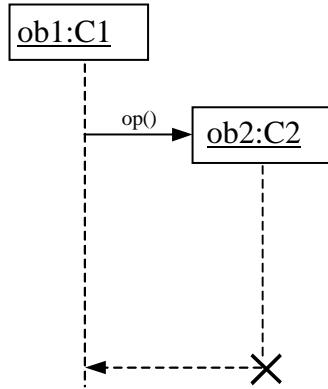
Figure 9:   *Example Sequence Diagram*

Instances have a "lifeline" drawn as a vertical line along the time axis. A lifeline can exist to describe that the particular instance already exists (e.g., instance ob1 of type C1). A lifeline can begin and end to show the creation and destruction of an instance, for example, instance ob2 of type C2. The lifeline starts at the box that shows the instance and ends at the big X. The arrow labelled *op()* depicts the message that creates the instance. A stimulus is shown as a horizontal arrow. The direction of the arrow defines the producer (start of the arrow) and the consumer (end of the arrow) of the stimulus. A stimulus can have a name, which describes the stimulus, and can map to a function (operation) of the instance that receives the stimulus. A stimulus can be drawn as a dotted line. In that case it describes a return of control to the sender of a stimulus. Different notations for arrows are used to represent different properties of stimuli. There are notations that distinguish between synchronous and asynchronous communication and timer stimuli, and between periodic and aperiodic events.

Usually only time sequences are important, but in real-time applications, the time axis could be an actual metric.

Figure 10 shows an example sequence diagram.
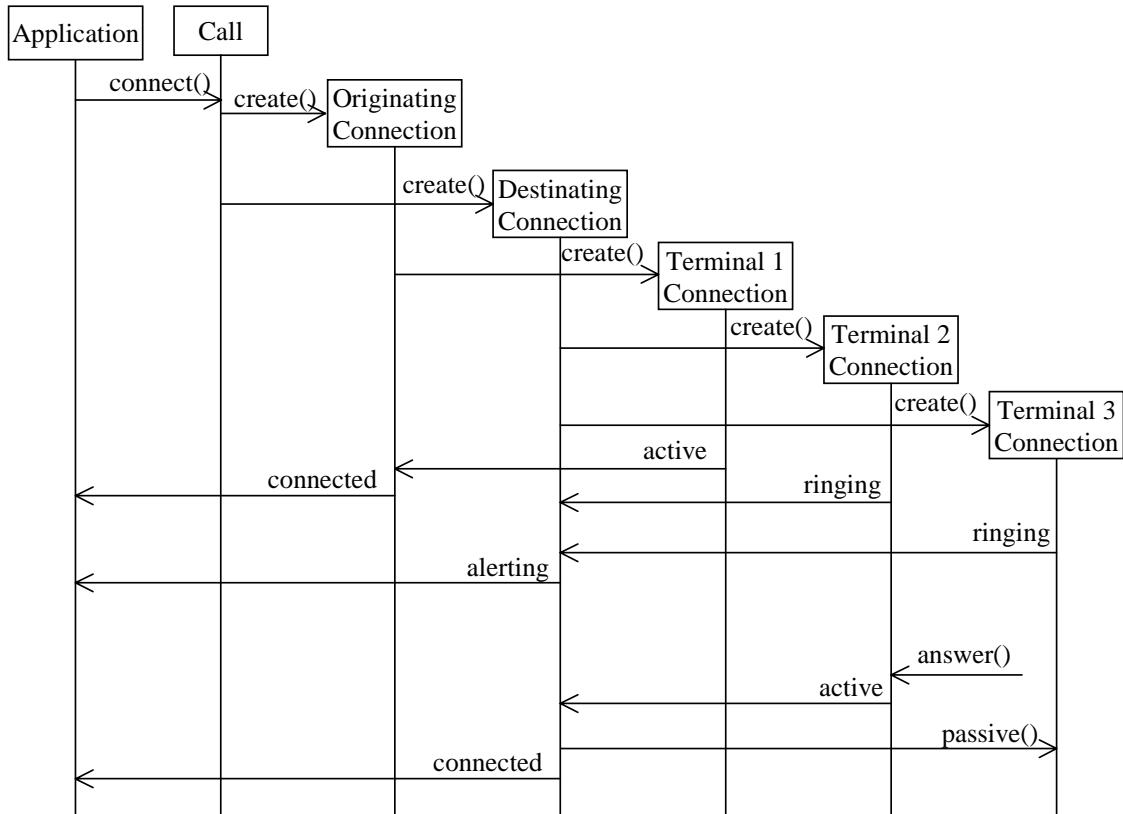
Figure 10: Example Sequence Diagram of "Establish Point-to-Point Connection"

The lifeline is shown as a vertical line to indicate the period in which the instance is active.
The vertical ordering of stimuli shows the ordering in time. Vertical distances between stimuli
may describe time duration in the sense that a greater distance stands for a longer time.

Figure 11 shows an example of a procedural sequence diagram.
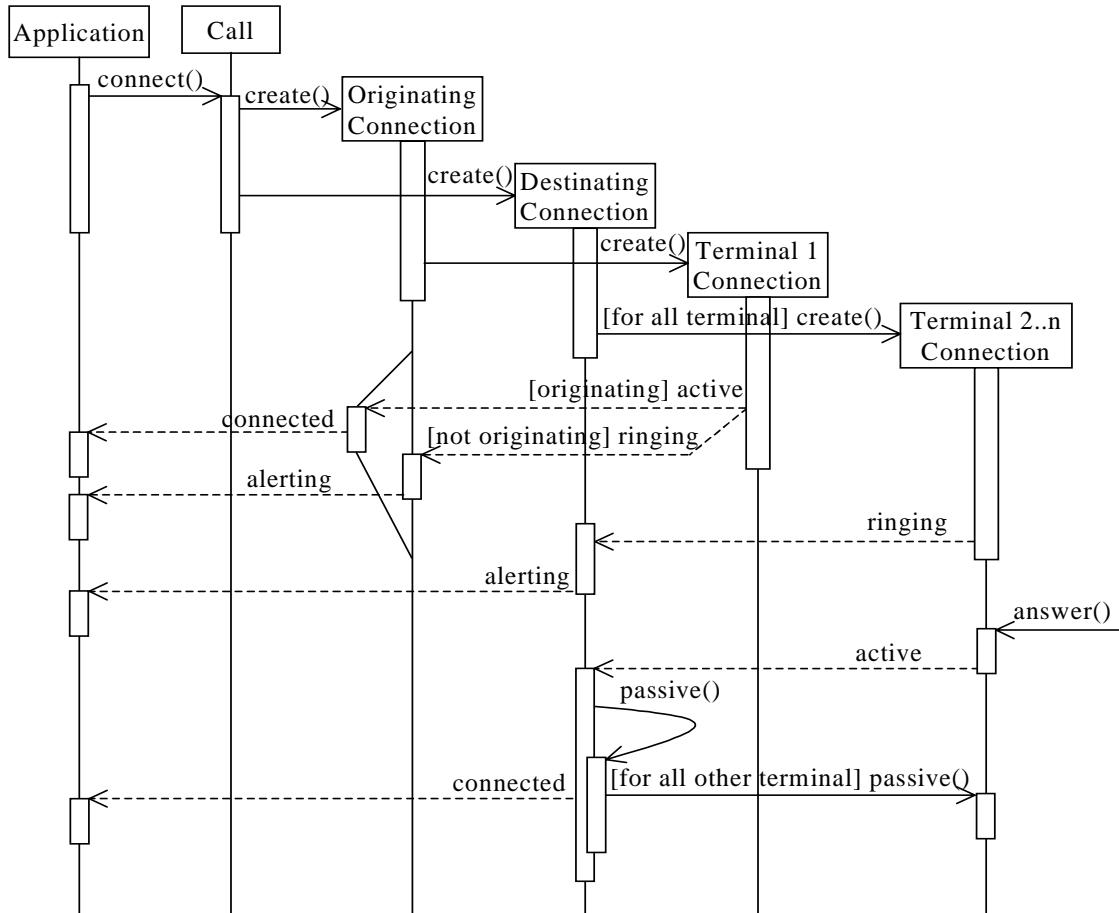
Figure 11: Procedural Sequence Diagram of "Establish Point-to-Point Connection"

An arrow (solid line) maps into a stimulus triggering a synchronous action, which is normally a function or method call. A "focus of control" (thin boxes over the lifeline of an instance) is added in this diagram style to show that some computation is done by the instance. The arrowhead pointing to a focus of control activates this function. Alternative execution paths as shown for the "Originating Connection" instance as well as possible parallel execution paths as shown for the "passive()" function of the "Destinating Connection" instance can be represented. Arrows with dotted lines represent asynchronous events that trigger activities in the instance to which the arrowhead points.

Although now a richer notation is available, not all possible concurrency can be shown in this style of sequence diagram. For example, the function of the instance "Destinating Connection" triggered by the event "active" could spawn concurrent threads that executes the "passive()" function of the same instance in parallel. The diagram is not specific at this point. The way it shows the behavior would allow for parallel as well as sequential execution.

A constraint language (such as the Object Constraint Language [OCL] described by the Object Management Group [OMG 01]) can be used in order to add more precise definitions of condi-

tions like guard or iteration conditions. OCL statements can be attached to the arrow and become *recurrence* values of the action attached to the stimulus. A return arrow departing the end of the focus of control maps into a stimulus that (re)activates the sender of the predecessor stimulus.

6.2.4   Collaboration Diagrams

Collaboration diagrams are component oriented. They show the relationships among the interfaces (normally call interfaces) of instances and are better for understanding all of the effects on a given instance and for procedural design. In particular, a collaboration diagram shows the instances participating in an interaction that exchange stimuli to accomplish a purpose. Instances shown in a collaboration diagram are those of elements described in the accompanying structural representation and show the aspects of the structural elements that are affected by the interaction. In fact, an instance shown in the collaboration diagram may represent only parts of the according structural element.

Collaboration diagrams are very useful when the task is to verify that a structure design can fulfill the functional requirements. They are not very useful if the understanding of concurrent actions is important, for example in a performance analysis.

For example in the structural description, there might be an element that stands for a bank account. In a collaboration diagram that shows what happens in a banking system if a user withdraws some money, only the money-manipulating aspect of a bank account is required and shown. In addition to this, the structural description about a bank account may also include maintenance features such as changing the address of this account's owner. The behavior of this feature is not important when describing the behavior of a withdrawal. However, there might be another collaboration diagram that describes the behavior of the bank system when an owner's address needs to be changed. In both diagrams, instances of a bank account will be shown, but both instances only show the particular aspects that are important for the specific diagram.

A collaboration diagram also shows the relationships among the instances, called links. Links show the important aspects of the relationships between those structural instances. Links between the same instances in different collaboration diagrams can show different aspects of relationships between the according structural elements. Links between instances have no direction. A link only states that the connected instances can interact with each other. If a more accurate definition is required, additional representational elements (perhaps a textual description) have to be introduced.

Sequence diagrams and collaboration diagrams express similar information. Some people prefer the sequence diagram because it shows time sequences explicitly, making it easy to see the order in which things occur. (Collaboration diagrams indicate sequencing using numbers.) Other people prefer the collaboration diagram because it shows element relationships, making it easy to see how elements are statically connected. (Sequence diagrams do not show these relationships.) Figure 12 shows an example of a collaboration diagram.

:Application

1: connect()

:Call

1.1a: create()

1.1a.1.1.1:
connected

2.1.2:
connected

1.1b: create()

1.1b.1.1.1:
alerting

Originating
:Connection

Destinating
:Connection

1.1b.1a: create()

1.1b.1b: create()

2.1.1:
passive()

1.1a.1.1: active

1.1a.1: create()

1.1b.1a.1:
ringing

2.1: active

1.1b.1b.1:
ringing

Terminal1
:Terminal Connection

Terminal2
:Terminal Connection

Terminal3
:Terminal Connection

2: answer()
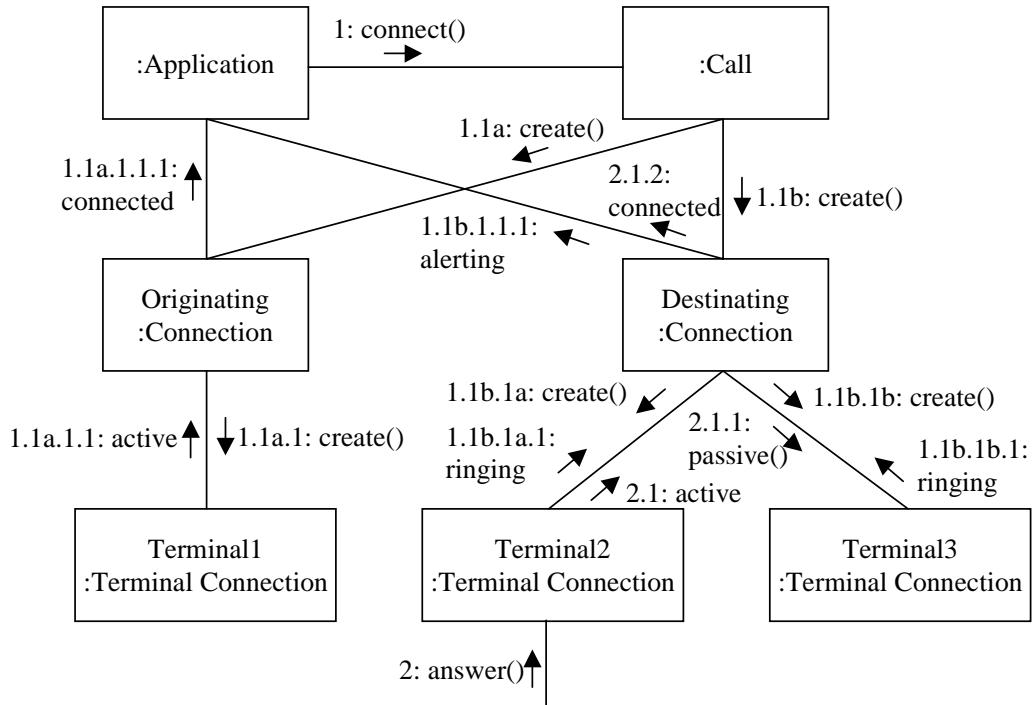
Figure 12: Example Collaboration Diagram of "Establish Point-to-Point Connection"

The sequence of stimuli are shown as little arrows attached to a link between the instances. The direction of the arrow defines the sender and receiver of the stimulus. Special types of arrows (such as half-headed arrows) can be used to depict different kinds of communication such as asynchronous, synchronous, and time-out. Sequence numbers can be added to stimuli to show which stimulus follows which. Sub-numbering can be used to show nested stimuli and/or parallelism. For example, the stimulus with a sequence number 1.1a is the first stimulus sent as a result of receiving stimulus number 1. The letter *a* at the end means that there is another stimulus (1.1b) that can be performed in parallel. This numbering scheme may be useful for showing sequences and parallelism, but it tends to make a diagram unreadable.

6.2.5 MSCs

An MSC is a message-oriented representation that contains the description of the asynchronous communication between instances. Simple MSCs almost look like sequence diagrams, but they have a more specific definition and a richer notation. The main area of application for MSCs is as an overview specification of the communication behavior among interacting systems, in particular telecommunication switching systems.

MSCs may be used for: requirement specification, simulation, and validation; test-case specification; and the documentation of systems. They provide a description of traces through the system in the form of a message flow. A big advantage of MSCs is that in addition to graphical representations, they have a textual specification language defined for them. This allows a

more formalized specification with the ability to generate test cases that test an implementation against the specification.

MSCs can often be seen in conjunction with the SDL. Both the SDL and the MSC language were defined and standardized by the ITU. While MSCs, as shown, focus to represent the message exchange *between* instances (systems, processes, etc.), the SDL was defined to describe what happens (or should happen) *in* a system or process. In that respect MSCs and SDL charts complement each other.

Though MSCs look similar to sequence diagrams, they are used for different purposes. A sequence diagram shows which parties are involved and how, and is system centric in that it is used to track a scenario through the system. MSCs are element centric, focusing on the element and how it interacts with its environment without regard to the identity of other elements.

The most fundamental language constructs of MSCs are instances and messages describing communication events. The example shown in Figure 13 shows how a JavaPhone application interacts with the JavaPhone layer in order to establish a Point-to-Point Connection. In an MSC, communication with outside elements is shown by message flow to and from the frame that marks the system environment. The example also shows descriptions of actions (Alert and Establish Connection) as well as the setting and resetting of a timer.

The complete MSC language has primitives for local actions, timers (set, reset, and time-out), process creation, process stop, and so forth. Furthermore MSCs have a means to show decomposition and so can be used to construct modular specifications.
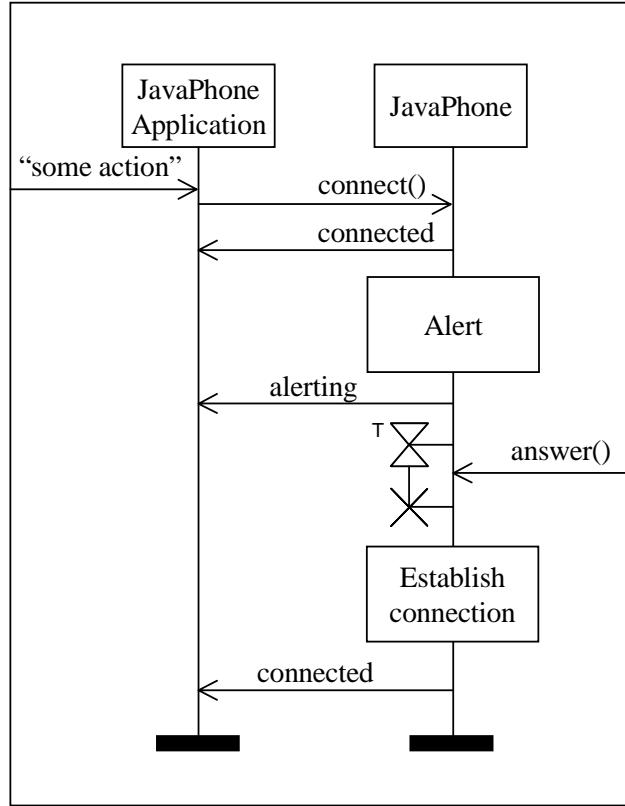

Figure 13: An Example of an MSC

Instances are shown as a box with a vertical line. The message flow is presented by arrows, which may be horizontal, or with a downward slope with respect to the direction of the arrow to indicate the flow of time. In addition, the horizontal arrow lines may be bent to admit message crossing. The head of the message arrow indicates message consumption, while the opposite end indicates message sending. Along each instance axis, a total ordering of the described communication events is assumed. Events of different instances are ordered only via messages, since a message must be sent before it is consumed. Within an MSC, the system environment is graphically represented by a frame, which forms the boundary of the diagram. Communication arrows to and from the frame show message exchange with elements outside the scope of the diagram.

7   Summary

Table 2 summarizes the major features of the notations described in this report.

Table 2:   Features Supported by the Different Representation Techniques

Notation	Class	Focus	Stimulus	Activity	Component	Timing
Collaboration diagram	Trace	Component	+	-	+	-
MSC	Trace	Message	+	+	0	0
Procedural sequence diagram	Trace	Component	+	+	+	+
ROOMchart	Static	State	+	0	-	+
SDL	Static	Transition	0	+	-	0
Sequence diagram	Trace	Message	+	0	+	+
Statechart	Static	State	+	0	-	0
UCM	Trace	Flow	0	0	+	-
Use Case	Trace	Activity	0	0	-	-
Z	Static	Activity	-	+	-	-

+ (plus sign) Means that the representation fully supports this feature
0 (zero) Means that the feature is somehow supported by the representation, yet there
   are other representations that are more appropriate if the understanding of a design
   depends on this feature
- (minus sign) Means that the representation does not or only very weakly supports a
   feature

8   For Further Reading

A rich source for behavior descriptions can be found in the UML definition that is publicly available from the OMG. On the OMG Web site, you can find definitions, descriptions, and examples of sequence and collaboration diagrams as well as example use cases and statecharts [OMG]. Several books also explain the UML and its usage in detail. Two seminal books that are valuable references are *The Unified Modeling Language User Guide* [Booch et al. 99b] and *The Unified Software Development Process* [Booch et al. 99a].

Books that serve as practical guides for using both ROOMcharts and statecharts include *Real-Time Object-Oriented Modeling* [Selic et al. 94] and *Modeling Reactive Systems With Statecharts: The Statemate Approach* [Harel & Politi 98]. ROOM has been incorporated into Rational UML tools.

MSCs, especially when combined with SDL diagrams, are broadly used by the telecommunication industry. Both languages are standardized by the ITU. On the ITU Web site, you can find all the references to resources (such as documentation and tool vendors) that you'll need to understand and use MSCs and the SDL [ITU]. Additional information and pointers to events, tools, and papers can be found at the SDL Forum Society's Web site [SDL]. This society currently recommends *SDL Formal Object-Oriented Language for Communicating Systems* [Ellsberger et al. 97] as the best practical guide to using SDL.

Many books have been written about use cases. The book from Ivar Jacobson that started the whole use-case discussion, *Object-Oriented Software Engineering: A Case-Driven Approach* [Jacobson 92], can serve as a starting point to understanding what was originally meant by use cases and their underlying concepts.

UCMs are still being researched, but there is a user group that tries to show the value of UCMs by applying the method to several projects. You can find interesting information at this user group's Web site [UCM User Group], including a free download of the book *Use Case Maps for Object-Oriented Systems* [Buhr & Casselman 96] and access to a free tool that supports UCMs.

The Z language was originally developed at Oxford University in the late 70s and has been extended by a number of groups since then. A large number of support tools to help create and analyze specifications have been developed by various groups and are available freely over the internet. A great resource for information and pointers is the Web archive, <http://www.afm.sbu.ac.uk/z>. There are a number of books that are available through most bookstores to help you use the Z language. *The Z Notation: A Reference Manual, 2nd Ed.* [Spivey 88a] provides a good reference in terms of a standard set of features.

Other notations are emerging but not widely used. Some are domain specific like MetaH, and others are more general like Rapide. Rapide has been designed to support the development of large, perhaps distributed, component-based systems [Augustin et al. 95]. Rapide descriptions are stated in a textual format that can be translated into a box-and-arrow diagram of a set of

connected components. System descriptions are composed of type specifications for component interfaces and architecture specifications for permissible connections among a system's components. Rapide is an event-based simulation language that provides support for the dynamic addition and deletion of pre-declared components based on the observation of specified patterns of events during the system's execution.

The Rapide tool set includes a graphical design environment that allows a designer to describe and simulate a system. The result of a Rapide simulation is a *POSET*, a partially ordered set of events that forms a trace of the system's execution. The simulation and analysis tools support exploring the correctness and completeness of the architecture. Rapide supports the use of two clocks and synchronous as well as asynchronous communication. A good tutorial along with other information and manuals associated with Rapide are available from the Rapide Web site [Rapide]. Other publications containing information on specific aspects of Rapide are listed in "References" on page 35 [Augustin et al. 95, Luckham & Vera 95, Perrochon & Mann 99].

MetaH was designed specifically to support the development of real-time, fault-tolerant systems. Its primary emphasis is on avionics applications, although it has also been used to describe a variety of system types. MetaH can be used in combination with ControlH, which is used to document and analyze hardware systems. When those two are used in combination, the system supports the analysis of stability, performance, robustness, schedulability, reliability, and security.

The style of specification is iterative, beginning with partial specifications based on system requirements and continuing to lower levels of refinement in the form of source objects. MetaH has capabilities that support the hierarchical specification of both software and hardware components, and the automatic generation of the "glue code" to combine predefined software and hardware components into a complete application. A user manual, instructions for obtaining an evaluation copy of the tool for use on Windows NT version 4.0, and other associated information about MetaH is available at the MetaH Web site [Honeywell]. Honeywell also has a document on its Web site that describes both ControlH and MetaH in terms of their relationship to domain-specific software architecture [Vestal 94]. Additional publications about MetaH[2] are listed in "References" on page 35 [Colbert et al. 00, Feiler et al. 00, Honeywell 00].

Architecture description languages (ADLs) have been developed within the research community to support the description, in textual form, of both the structure and the behavior of software systems. Stafford and Wolf discuss ADLs and provide a table containing references to and brief descriptions of several languages [Stafford & Wolf 01].

[2.] Another publication which is not publicly available is "Software Portability Gains Realized with MetaH, an Avionics Architecture Description Language," from the Digital Avionics Systems Conference. This conference was held in St. Louis, Missouri on October 23-29, 1999. You can obtain this document from the author, Bruce A. Lewis.

8.1 Useful Web Sites

Table 3 lists the URLs you can access on the Web if you need more information about the components discussed in this document.

Table 3:   URLs to Go to for More Information

For information on:	See this Web site:
collaboration diagrams	<http://www.omg.org/uml>
MetaH	<http://www.htc.honeywell.com/metah> <http://www.htc.honeywell.com/projects/dssa/dssa_tools /dssa_tools_mhch.html>
MSCs	<http://www.itu.int/home/index.html> <http://www.sdl-forum.org>
Rapide	<http://pavg.stanford.edu/rapide> <http://pavg.stanford.edu/rapide/examples/teaching/dtp /index.html>
SDL	<http://www.sdl-forum.org> <http://www.itu.int/home/index.html>
sequence diagrams	<http://www.omg.org/uml>
statecharts	<http://www.itu.int/home/index.html>
UCMs	<http://www.usecasemaps.org>
Z	<http://spivey.oriel.ox.ac.uk/~mike/zrm> <http://www.afm.sbu.ac.uk/z/>

References

[Augustin et al. 95]   Augustin, L. M.; Luckham, D. C.; Kenney, J. J.; Mann, W.; & Vera, D. Bryan. "Specification and Analysis of System Architecture Using Rapide." *Transactions on Software Engineering 21*, 4 (April 1995): 336-355.

[Bass et al. 98]   Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

[Bachmann et al. 01]   Bachmann, F.; Bass, L.; Clements, P.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Organization of Documentation Package* (CMU/SEI-2001-TN-010). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <http://www.sei.cmu.edu/publications /documents/01.reports/01tn010.html>.

[Bachmann et al. 00]   Bachmann, F.; Bass, L.; Carriere, J.; Clements, P.; Garlan, D.; Ivers, J.; Nord, R.; & Little, R. *Software Architecture Documentation in Practice: Documenting Architectural Layers* (CMU/SEI-2000-SR-004, ADA377988). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports /00sr004.html>.

[Booch et al. 99a]   Booch, G.; Jacobson, I.; & Rumbaugh, J. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

[Booch et al. 99b]   Booch, G.; Jacobson, I.; & Rumbaugh, J. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[Buhr & Casselman 96]   Buhr, R. J. A. & Casselman, R. S. *Use Case Maps for Object-Oriented Systems*. Upper Saddle River, NJ: Prentice Hall, 1996.

[Colbert et al. 00]	Colbert, E.; Lewis, B.; & Vestal, S. "Developing Evolvable, Embedded, Time-Critical Systems with MetaH," 447-456. *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems.* Santa Barbara, California, July 30 - August 4, 2000. Los Alamitos, CA: IEEE Computer Society, 2000.
[Ellsberger et al. 97]	Ellsberger, J.; Hogrefe, D.; & Sarma, A. *SDL: Formal Object-Oriented Language for Communicating Systems.* New York, NY: Prentice Hall Europe, 1997.
[Feiler et al. 00]	Feiler, P.; Lewis, B.; & Vestal, S. *Improving Predictability in Embedded, Real-Time Systems* (CMU/SEI-2000-SR-011, ADA387262). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu /publications/documents/00.reports/00sr011.html>.
[Harel & Politi 98]	Harel, D. & Politi, M. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* New York, NY: McGraw-Hill, 1998.
[Harel 87]	Harel, D. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming 8,* 3 (June 1987): 231-274.
[Honeywell 00]	Honeywell Laboratories. *MetaH User's Guide* [online]. <http://www.htc.honeywell.com/metah/uguide.pdf> (2000).
[Honeywell]	Honeywell Laboratories, MetaH Evaluation and Support Site. <http://www.htc.honeywell.com/metah>.
[ITU]	International Telecommunication Union. <http://www.itu.int /home/index.html>.
[Jacobson 92]	Jacobson, I. *Object-Oriented Software Engineering: A Case-Driven Approach.* Reading, MA: Addison-Wesley, 1992.
[Kazman & Klein 99]	Kazman, R. & Klein, M. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022, ADA371802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. <http://www.sei.cmu.edu/publications/documents/99.reports /99tr022/99tr022abstract.html>.

[Luckham & Vera 95]	Luckham, D. C. & Vera, J. "An Event-Based Architecture Definition Language." *Transactions on Software Engineering 21*, 9 (September 1995): 717-734.
[OMG]	Object Management Group. <http://www.omg.org/uml>.
[OMG 01]	Object Management Group. *OMG Unified Modeling Language Specification Version 1.4 (draft)* [online]. <http://www.omg.org/docs/ad/01-02-13.pdf> (2001).
[Perrochon & Mann 99]	Perrochon, L. & Mann, W. "Inferred Designs." *IEEE Software 16*, 5 (September/October 1999): 46-51.
[Rapide]	The Stanford Rapide Project. <http://pavg.stanford.edu/rapide>.
[Rosenberg & Scott 99]	Rosenberg, D. & Scott, K. *Use-Case-Driven Object Modeling with UML: A Practical Approach*. Reading, MA: Addison-Wesley, 1999.
[SDL]	SDL Forum Society. <http://www.sdl-forum.org>.
[Selic et al. 94]	Selic, B.; Gullekson, G.; & Ward, P. T. *Real-Time Object-Oriented Modeling*. New York, NY: John Wiley, 1994.
[Sowmya & Ramesh 98]	Sowmya, A. & Ramesh, S. "Extending Statecharts with Temporal Logic." *Transactions on Software Engineering 24*, 3 (March 1998): 216-231.
[Spitznagel & Garlan 98]	Spitznagel, B. & Garlan, D. "Architecture-Based Performance Analysis," 146-151. *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*. San Francisco, California, June 18-20, 1998. Skokie, IL: Knowledge Systems Institute, 1998.
[Spivey 88a]	Spivey, J. M. *The Z Notation: A Reference Manual* [online]. <http://spivey.oriel.ox.ac.uk/~mike/zrm> (1988).
[Spivey 88b]	Spivey, J. M. *Understanding Z: A Specification Language and Its Formal Semantics*. New York, NY: Cambridge University Press, 1988.

[Stafford & Wolf 01]   Stafford, J. A. & Wolf, A. L. Ch. 20, "Software Architecture," 371-387. *Component-Based Software Engineering: Putting the Pieces Together.* Heineman, G. T. & Councill, W. T., eds. Boston, MA: Addison-Wesley, 2001.

[Stafford & Wolf 00]   Stafford, J. A. & Wolf, A. L. "Annotating Components to Support Component-Based Static Analyses of Software Systems" [CD-ROM]. *Proceedings of the Grace Hopper Celebration of Women in Computing Conference 2000.* Hyannis, Massachusetts, September 14-16, 2000. Palo Alto, CA: Institute for Women in Technology, 2000. Also published as report CU-CS-896-99 [online]. Boulder, CO: University of Colorado, Department of Computer Science, 1999. <http://www.cs.colorado.edu/department/publications/reports/Judith_A._Stafford.html>.

[UCM User Group]   Use-Case Maps User Group. <http://www.usecasemaps.org>.

[Vestal 94]   Vestal, S. "Mappings Between ControlH and MetaH" [online]. <http://www.htc.honeywell.com:80/projects/dssa/dssa_tools/dssa_tools_mhch.html> (1994).

1. AGENCY USE ONLY (leave blank)		2. REPORT DATE January 2002	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Documenting Software Architecture: Documenting Behavior			5. FUNDING NUMBERS C — F19628-00-C-0003
6. AUTHOR(S) Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2002-TN-001
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12.b DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

This report represents another milestone of a work in progress: a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively titled *Documenting Software Architectures*, will be published in early 2002 by Addison-Wesley as part of the Software Engineering Institute (SEI) Series on Software Engineering.

The book is intended to address a lack of language-independent guidance about how to capture an architecture in a written form that can provide a unified design vision to all of the stakeholders on a development project.

A central precept of the book is that documenting an architecture entails two essential steps: 1) documenting the set of relevant views of that architecture and then completing the picture by 2) documenting information that transcends any single view. The book's audience is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

This technical note describes ways to document an important but often overlooked aspect of software architecture: the behavior of systems, subsystems, and components.

14. SUBJECT TERMS software architecture, documentation, architectural views, Z, statechart, SDL, behavior			15. NUMBER OF PAGES 50
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL